A Variability Management Process for Software Product Lines

Edson Alves de Oliveira Junior, Itana M. S. Gimenes, Elisa Hatsue Moriya Huzita

Universidade Estadual de Maringá (UEM)

Departamento de Informática

José Carlos Maldonado

Universidade de São Paulo (USP)

Departamento de Ciências de Computação e Estatística

Abstract

The software product line approach (PL) promotes the generation of specific products from a set of core assets for a given domain. This approach is applicable to domains in which products have well-defined commonalities and variation points. Variability management is concerned with the management of the differences between products throughout the PL lifecycle. This paper presents a UML-based process for variability management that allows identification, representation and delimitation of variabilities as well as identification of mechanisms for variability implementation. The process is illustrated with excerpts of a case study carried out within the context of an existing PL for the Workflow Management System (WfMS) domain. The case study was carried out based on the experimental software engineering concepts. The results have shown that the proposed process has made explicit a higher number of variabilities than does the existing PL process, and it offers better support for variability tracing.

1. Introduction

The software product line (PL) approach aims at promoting the generation of specific products of a domain (product family) based on the reuse of a well-defined infrastructure, called *core asset* [1].

The benefits obtained with a PL approach include [1, 2]: better understanding of the domains, more artifact reuse, and less time to market. Practical evidence of these benefits can be seen from organization reports such as Nokia [3]. As a result, there are several efforts, both academic and industrial, to reduce the difficulties in the PL adoption [4].

The core asset is the main part of the PL; it contains the architecture of the PL, and its components are represented in a way that makes clear the common and variable aspects of the potential products of the PL. The ability and simplicity of producing products from a PL depend on how well its core asset is designed. The more generic are the artifacts of the core asset, the more products can be generated from a PL. This generality requires the postponing of design decisions [1, 5] that allow the discrimination of products. These design decision issues are treated as *variabilities*.

Although variability management is recognized as an important issue for the success of PLs, there are not many solutions available in the literature [2]. In addition, each existing solution is applied to only a specific PL approach. Thus, there is a lack of an overall reasoning about variability management.

This paper presents a process for variability management which includes tasks for identification, representation, delimitation of variabilities; and identification of implementation mechanisms of variabilities. The process supports the whole PL lifecycle. In addition, it supports variability tracing and analysis of the configuration of specific products.

The process was conceived within the context of an existing component-based product line for Workflow Management Systems (WfMS) [6], [7]. The evaluation of the process was carried out as a case study which followed the concepts of experimental software engineering.

This paper is organized as follows: Section 2 presents a discussion about important issues of variability management. Section 3 presents the proposed process. Sections 4 and 5 present lessons learned and related work. Finally, Section 6 presents the conclusions.

2. Variability Management in Software Product Line

This section introduces the main concepts and issues involved in variability management.

Variability is the general term used to refer to the variable aspects of the products of a PL. It is described through variation points and variants. A variation point is the specific place in a PL artifact to which a design decision is connected. Each variation point is associated with a set of variants that corresponds to design alternatives to resolve the variability [2].

Variability management includes issues such as: (i) variability identification and representation; (ii) variability binding, and (iii) variability control.

According to van Gurp, Bosch and Svahnberg [5], a variability can be initially identified based on the concept of feature. They define a feature as a logical unit of behavior that corresponds to a set of functional and quality requirements. This concept comes from domain engineering [8] and has been constantly improved to fit the demands of the product line approach [5, 9, 10, 11].

Features are usually represented through feature diagrams which are decorated trees that contain the features identified for a system family [10, 11]. Important attributes usually represented in a feature diagram are: the relationship between features and variation points, the relationship amongst features, and the feature binding time.

There are PL approaches that are purely based on the feature model and Domain Specific Languages (DSL) [12, 13]. Another important set of PL approaches maps the features to UML models [14, 15, 16] and provides UML extensions to represent variability along the PL lifecycle. The variability management process proposed in this paper is based on UML.

Jacobson, Griss and Jonsson [17] propose the currently most popular notation to represent variability in use cases. A blob ("•") is associated to the use case indicating that there are variable aspects associated with it. However, variabilities are not treated at the actor level. Morisio, Travassos and Stark [18] introduce a notation for class diagram; in this case the stereotypes "<<V>>" and "<<xorV>>" are associated to classes and methods. In this work, variability is represented by introducing stereotypes to the UML models: use case, class and component. In addition, UML notes are used to represent information regarding variation points and variants.

The variability binding time indicates the PL lifecycle milestone in which one of the variants associated with the variation point will be chosen [19]. Variability binding constrains also the choice of implementation mechanisms [20]. We adopt the classification of binding time proposed by Anastasopoulos and Gracek [21] which is: (i) compile time – the variability is resolved either before the program is compiled or at compile time; (ii) link time – the variability is resolved during module or library linking, by selecting different library with different versions of exported operations; (iii) runtime – the variability is resolved during program execution; (iv) update time or post runtime – the variability is resolved during the program updates or after its execution.

According to Fritsch, Lehn and Strohm [20] and Becker [22], variability management is related to every activity of a PL core asset development process. Van Gurp, Bosch and Svahnberg [5] suggest that the variability management process is composed of the following activities:

- Variability identification – consists of identifying the product differences and their location within the PL artifacts.
- Variability delimitation – defines the binding time and multiplicity.
- Variability implementation – is the selection of implementation mechanisms.
- Variant management – controls the variants and variation points.

These activities were used as a basis to conceive our process.

3. The Variability Management Process

This section presents the overall variability management process, its activities and its relationship with the PL development process.

Figure 1 presents the interaction between the core asset development process, represented by the activities vertically aligned on the left, and the variability management process, represented by the activities defined inside the right rectangle. The variability management process activities are executed by the PL manager. It is an iterative and incremental process that runs in parallel with the core asset development. After the execution of each activity of the core asset development, the variability management process is executed, thus progressively taking as input the output artifacts of the core asset development. As the activities are executed, the number of variabilities tends to increase. As the process is iterative, variability updates are allowed from any activity of the process.

The input and output artifacts of the activities are defined as follows. However, note that the input artifacts are made available according to the progress of the core asset development activities.

The proposed process consists of the following activities:

- Variability tracing definition, which takes the use case and the feature models as input and generates the variability tracing model as output;
- Variability identification, which takes the use case, the static type, and the feature models, plus the component model as input and generates the same artifacts with the variabilities identified as output;
- Variability delimitation, which takes the use case, the static type and the feature models, plus the component model as input and generates the same artifacts with the variabilities limited as output; and
- Identification of mechanisms for variability implementation, which takes the static type model and the component model as input and generates the variability implementation model as output.

In addition, the process is supported by a metadata model which describes the relationships among the PL artifacts. This model is described in Section 3.6.

The process consumes artifacts from the PL core asset as well as producing information for it. An example is the use case model and the static type model. They feed the variability management process and return to the core asset the variabilities identified and limited. However, there are models such as the variability tracing and implementation models that are originated in the variability management process.

The following subsections provide a summary of the case study undertaken to evaluate the proposed process and to describe the process activities.

3.1 Case Study

The case study was conceived based on the concepts of experimental software engineering and the evaluation of software engineering methods and tools [24, 25, 26]. An evaluation plan was developed consisting of the following activities: identification of the case study context, definition of hypotheses, selection of the pilot project, identification of comparison methods, planning of the case study, execution of the case study and analysis of results.

The context of the case study was an existing component-based product line for Workflow Management Systems (WfMS) [6], [7]. The design of the product line encompasses important concepts such as the central role of software architecture, frameworks and patterns. The fact that we represent all artifacts of the PL in UML makes it easier for designers of traditional approaches to understand our specification and also enable us to take profit of current support tools such as IBM/Rational Rose [23]. The objective of the case study was to re-specify the existing product line with the introduction of the proposed variability management process in order to observe the impact on the number of variabilities identified. Thus, the PL development activities were carried out interacting with the variability management activities, as established in the proposed process. The number of variabilities identified in the following models was measured: use case, static type and component. These numbers were than compared to the number of variabilities identified in the previous process.

The results confirmed that more variabilities were identified after the introduction of the variability management process (see Figure 6).

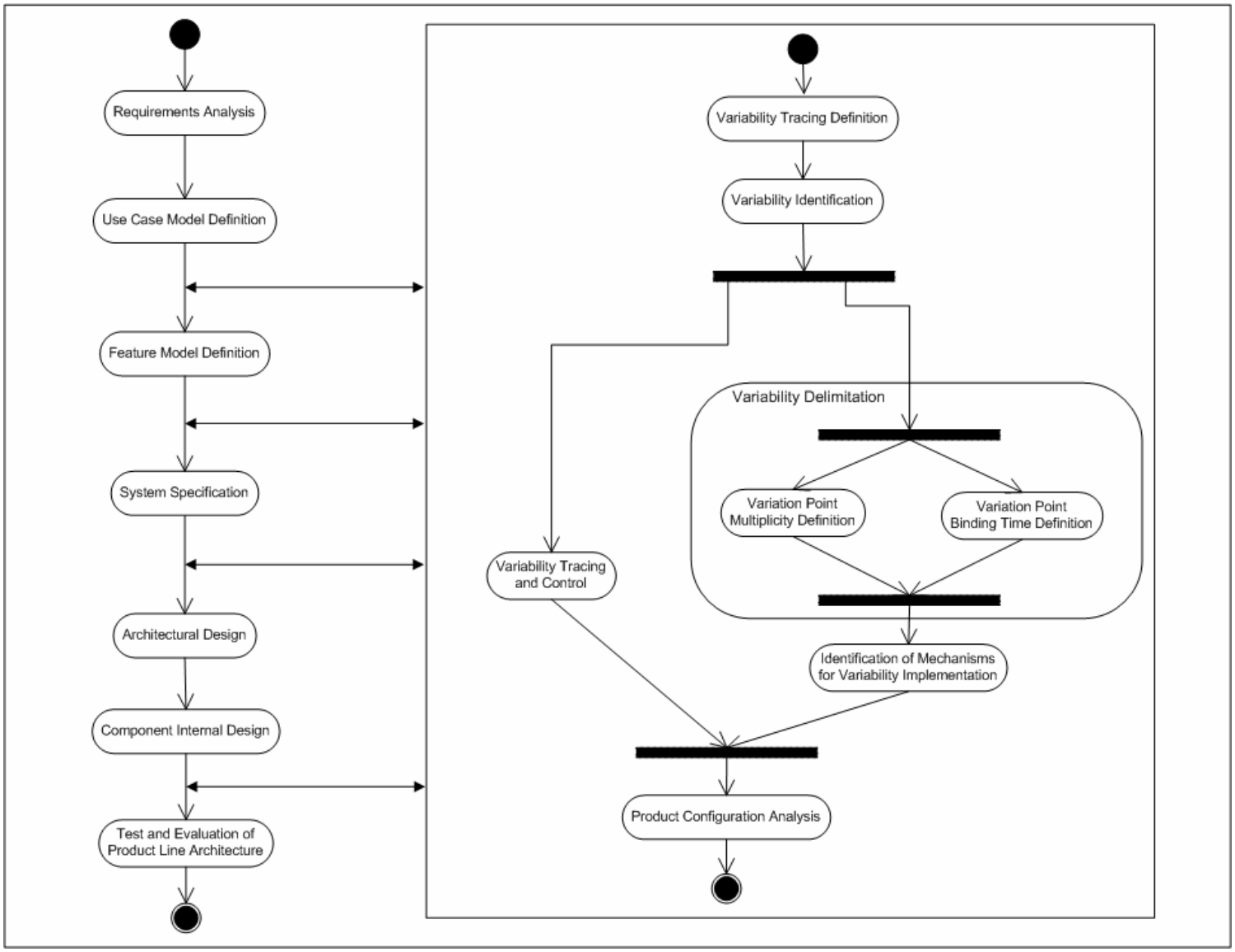

Figure 1: The variability management process and its interaction with the PL development process.

As support tool we used IBM/Rational Rose and Requisite Pro [23].

The activities described in the following sections are illustrated with an excerpt of the case study. For space reasons, the activities of identification and delimitation of variabilities are illustrated through the use case diagram. The representation of variabilities for other artifacts is similar.

3.2 Variability Tracing Definition

Variability tracing definition receives as input the use case model and the feature model built in the PL development process. The variability tracing model is built as follows:

- the features of the PL are listed;
- the use cases are listed;
- the relationship between feature and use cases are analyzed;
- crossing relations between use cases and features are marked with a blob.

This model is represented as a cross reference between features and use cases. For example, the feature `User Communication` is related to the use cases (Figure 2): `Communicate with Users`, `Communicate Via E-mail`, `Communicate Via Chat` and `Communicate Via Teleconference`, so the cross reference between them are marked. The model will support the tracing from the features to all UML models that are related to variable aspects throughout the lifecycle. Thus, if a certain feature needs to be removed from a given product, it is necessary to know the impact of the removal over related artifacts. The tracing is possible because the features are related to the use case model that is related to the static type model and the component model. For instance, if the feature `User Communication` is not selected for a certain product, the related use cases and its connected models (static type and component) have to be updated to reflect this.

The variability tracing model is based on the representation used for feature and use case tracing in the tool IBM/Rational Requisite Pro [23]. This tool allows indicating which use cases are related to the features and vice-versa.

3.3 Variability Identification

Variability identification receives as input the use case, feature, static type, and component models from the development activity in each interaction between the core asset development and the variability management process. It aims at progressively identifying the variability associated with the models. Appendix A presents the stereotypes used to represent variability on the artifacts of the PL. For each model, there are two columns. The first indicates the UML relation used to represent the type of variability between the variation point and its variants. The second indicates the stereotype used to indicate variability.

The stereotype `<<variationPoint>>` indicates that a model element represents a variation point and has associated variants. These variants can be represented with one of the following stereotypes: `<<mandatory>>`, indicating a compulsory variant; `<<optional>>`, indicating a variant that need not be chosen; `<<alternative_OR>>`, indicating that zero or more variants can be chosen; and, `<<alternative_XOR>>`, indicating that only one of the variants can be chosen.

The stereotypes `<<requires>>` and `<<mutex>>` are used to indicate dependency relationship between variants. The former indicates that once a source variant is chosen, also the target variants have to be chosen. In contrast, the latter indicates that once a source variant is chosen the target variants cannot be chosen.

UML notes are also used to support graphical representation of variabilities. A UML note represents essential information that allows answering questions such as:

- Which variants are related to a variation point?
- What is the binding time of the variants associated to a variation point?
- Is it possible to add new variants to the variant set associated with a given variation point?

The UML note associated with a variation point defines:

- the type of the relationship between the variation point and its variants, which are: {} indicating a mandatory or optional relation, {or} indicating an inclusive relation and {xor} indicating an exclusive relation;
- the name of the variation point;

- the multiplicity of the variation point, indicating the minimum number of variants to be chosen to resolve such variation point;
- the binding time of the variation point; and
- true or false indicating whether the variation point supports the addition of new variants to its associated set.

Figure 2 presents an example of variability identification in a use case model. In this Figure, the UML notes has three "?" because the variabilities were not limited yet.

Two variabilities were identified in this Figure. One is concerned with different forms of workflow execution, and the other is related to the kind of communication between workflow users. The former is represented by the use case Execute Workflow. The associated variants include Execute Workflow with Priority Control and Execute Workflow Serial. The kind of variability relationship is `<<alternative_XOR>>` because only one of the execution algorithms can be selected. The latter, is represented by the use case `Communicate with Users`. The associated variants are `Communicate Via Chat`, `Communicate Via E-mail` and `Communicate Via Teleconference`. The kind of variability relationship is `<<alternative_OR>>` as more than one kind of communication can be used.

Variability identification is a domain dependent activity which requires abilities of the PL managers and analysts. However, some guidelines may be offered, such as:

- elements of the use case model related with the stereotype `<<extend>>` or elements of the static type model related by inheritance suggest variation points with associated variants which are inclusive or exclusive alternative. For instance, in Figure 2, the use cases Execute Workflow with Priority Control and Execute Workflow Serial represent exclusive alternatives associated with the variation point Execute Workflow.
- elements of the artifacts related to the stereotype `<<include>>` in the use case model or to an association in the static type model suggest either mandatory or optional variation points. For instance, in Figure 2, the use case `Execute Script` represents a compulsory variant.

3.4 Variability Delimitation

Variability delimitation aims at defining the following attributes of a variation point: (i) multiplicity; (ii) binding time, and (iii) possibility, or not, of adding new elements to the associated variant set.

The multiplicity of a variation point indicates the minimum number of elements of the associated variant set that must be chosen to resolve the variability. The following rules are applied:

- a variation point with relation type optional has multiplicity 0 (zero), indicating that a variant can be chosen or not;
- a variation point with relation type mandatory has multiplicity 1 (one), indicating that the associated variants must be selected;
- a variation point with relation type exclusive alternative has multiplicity 1 (one), indicating that only one element of the possible set of variants must be selected; and
- a variation point with relation type inclusive alternative has multiplicity ranging from 0 (zero) to the maximum number of variants associated with the variation point, indicating that any number of variants of this interval can be chosen.

The binding time of a variation point, as defined in Section 2, can be as follows:

- Design – the variation point is resolved during the specification of the PL or of its products.
- Implementation – the variation point is resolved at programming time, before compilation.
- Compiling – the variation point is resolved during the compilation process.
- Linking – the variation point is resolved during the module or library linkage process.
- Runtime – the variation point is resolved at the PL product execution time.
- Updating – the variation point is resolved during the PL product update. An example is the inclusion of new module to in PL product when it is already running.

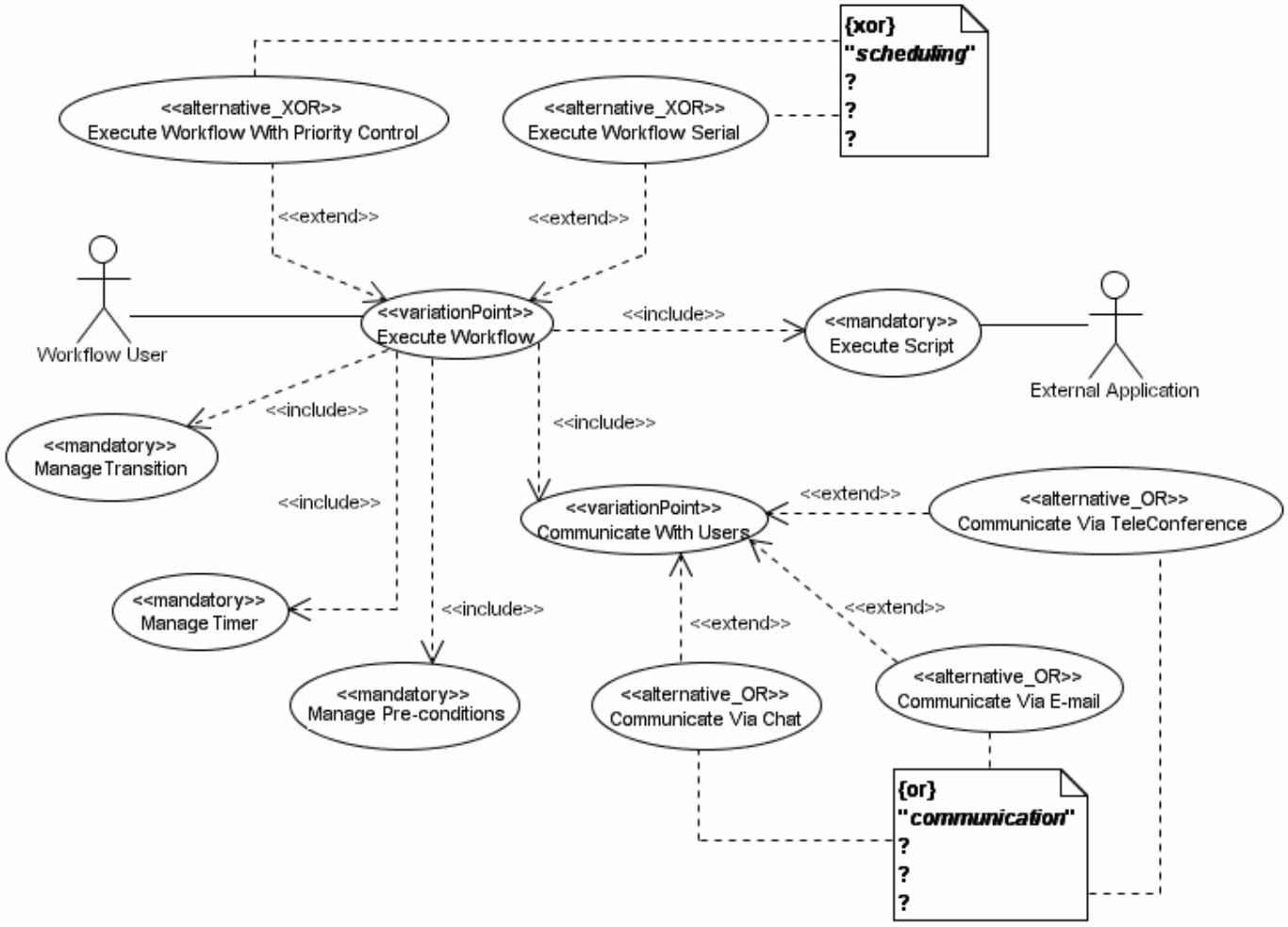

Figure 2: Variability identification in a use case model.

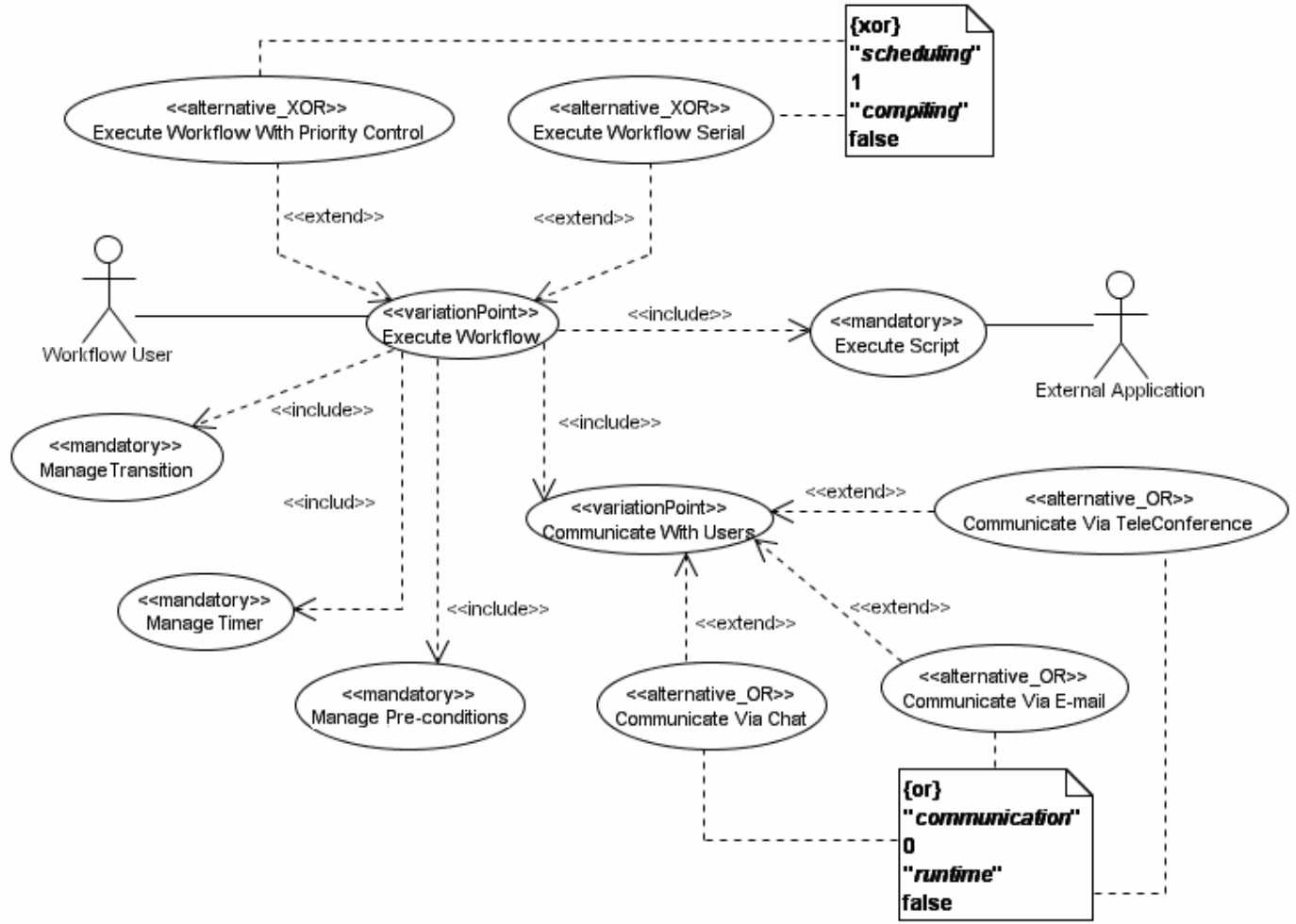

Figure 3: Representation of multiplicity and binding time of variation points in use case models.

The definition of the binding time is essential to determine the choice of implementation mechanisms, which are described in the next section. Figure 3 presents an example of variability delimitation in a use case model, for the variation points `Execute Workflow` and `Communicate with Users`, identified in Figure 2. The variability `"scheduling"` has multiplicity 1 as it requires the selection of only one of the variants: `Execute Workflow with Priority Control` or `Execute Workflow Serial`. The fourth line of the note indicates that the binding time is `"compiling"` whereas the fifth line is set to false to indicate that addition of new variants is not allowed. The note associated with the variability `"communication"` indicates that 0 (zero) or more variants can be selected; the binding time is `"runtime"`, and the addition of new variants is not allowed.

3.5 Identification of Mechanisms for Variability Implementation

The identification of mechanisms for variability implementation aims at selecting the mechanisms to be used to implement the variability. The input artifacts are the static type and the component models with their respective variabilities represented and limited, as a result of the previous activity.

The output artifact of this activity is an implementation model which is represented as a table. Each row of the table indicates the name of a variability, the artifact element in which it occurs, the binding time, the implementation mechanism and the implementation strategy. The model was built based on the variability implementation techniques proposed by Svahnberg, van Gurp and Bosch [35], Jacobson [17] and Anastasopoulos [21]. These techniques include inheritance, extension, and parameterization.

The implementation mechanism and strategy are chosen based on the binding time and the class or component in which the variability occurs.

As an example, Figure 4 presents the static type model for the WfMS PL. The class `TypeTool` represents a variation point and the classes `TypeInternalTool` and `TypeExternalTool` are the associated variants. The variability `"tool class type"` occurs at the class level and it is bound at compiling time. Thus, a possible implementation mechanism [35] is `"Code Fragment Superposition"`. The strategy `"override generic source code with specific one using aspect-oriented programming"` guides the variability implementation.

3.6 Variability Tracing and Control

Variability tracing and control aims at defining the relationship between artifacts in order to control variabilities. It is supported by the process metadata model, presented in Figure 5. This model describes the relationships among the PL artifacts. It was conceived based on the generic variability model proposed by Becker [22] to support variability management tools. The metamodel defines the relationships between variant artifacts of a PL and their variation points, variants, binding time and implementation mechanisms.

This metamodel together with the variability tracing model allows the association of a feature to the related use cases and therefore to the elements of the static type and component model. The execution of this activity consists of the instantiation of the metamodel for the PL.

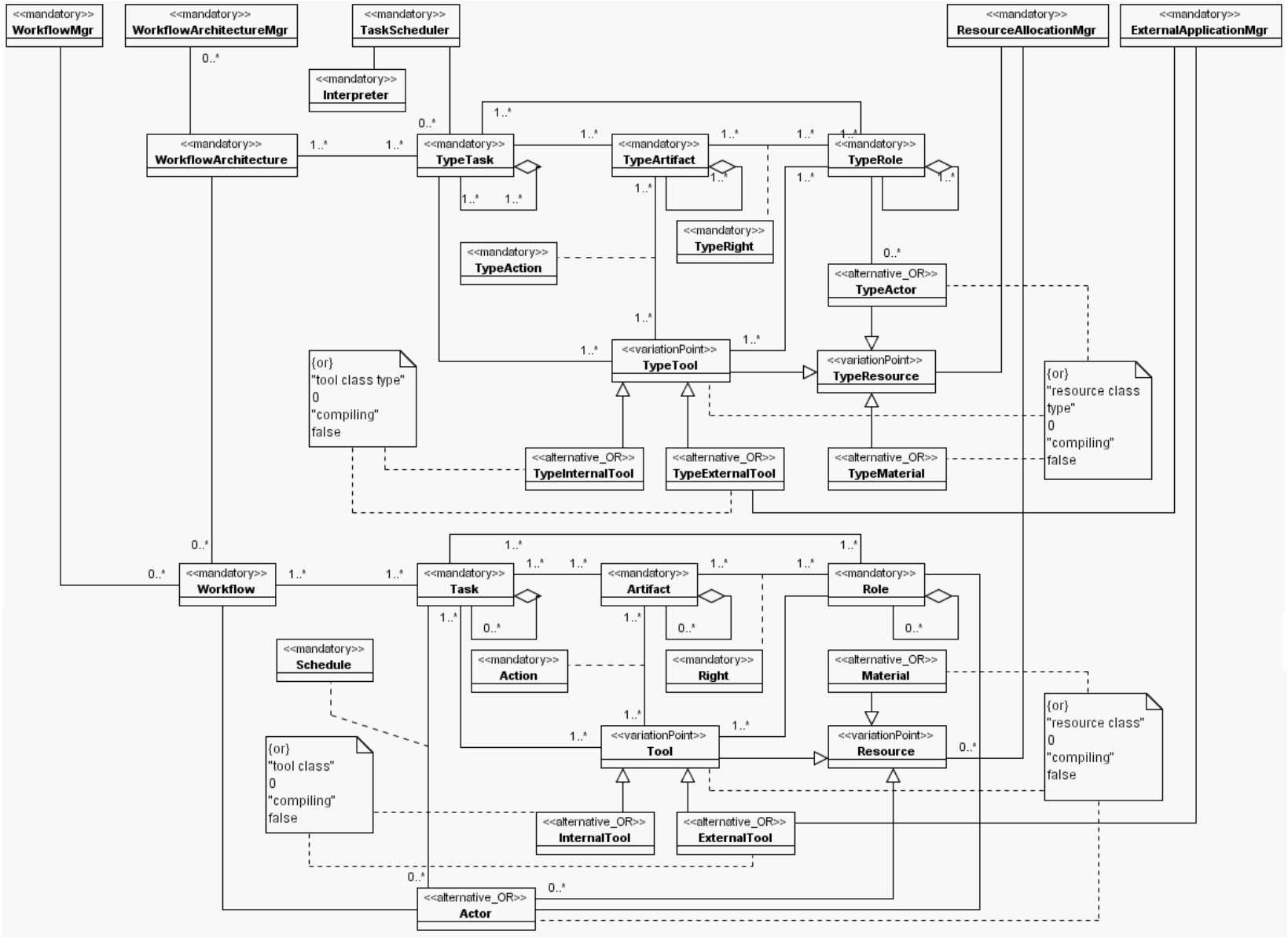

Figure 4: Static Type Model for WfMS.

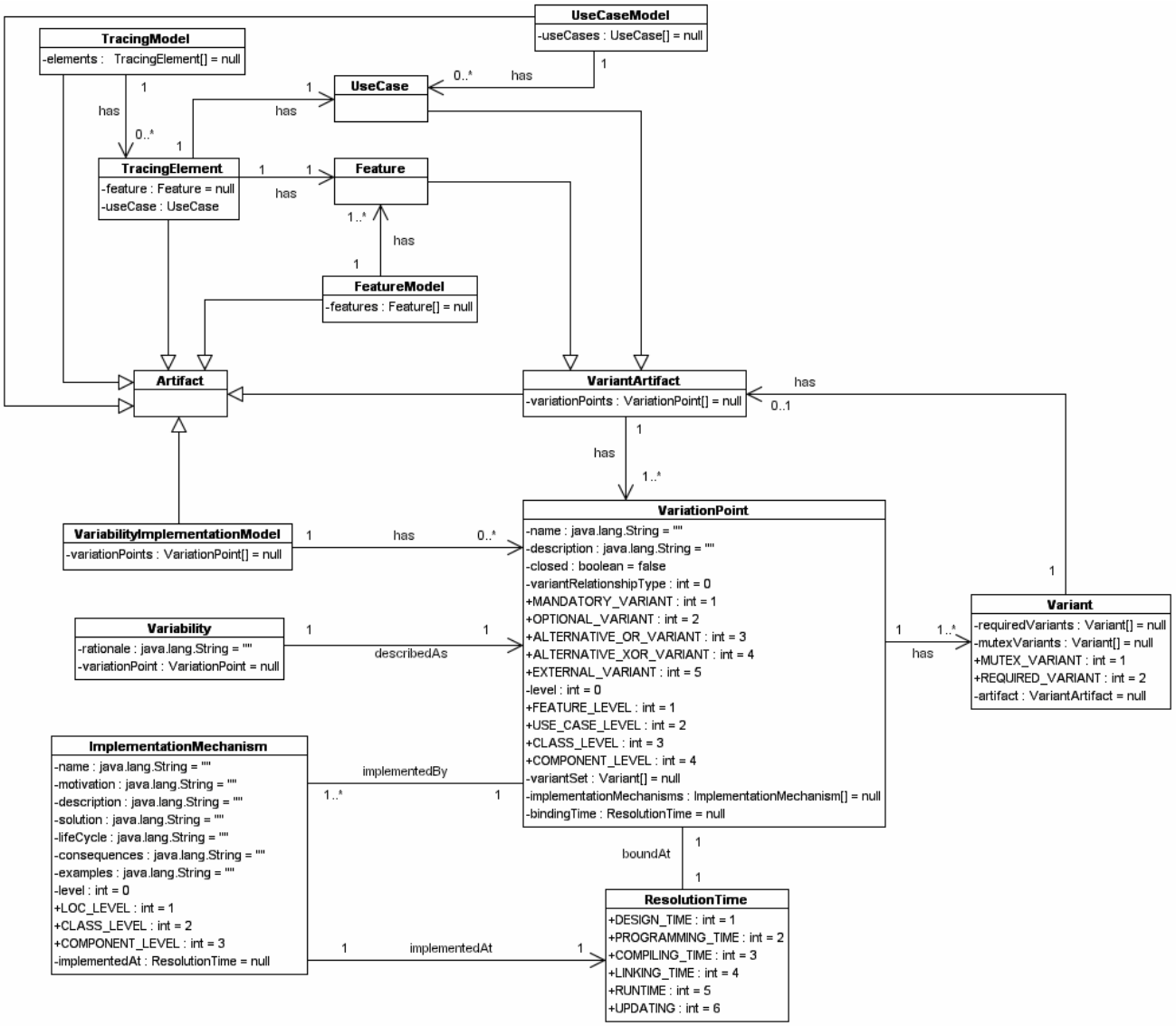

Figure 5: Metadata model to support the variability management process.

3.7 Configuration Analysis of Specific Products

Configuration analysis for specific products aims at investigating the impact of selecting possible features of the PL artifacts in order to analyse the feasibility of the production of a specific product which is a PL member. The selection of a feature may imply the selection of a product configuration according to the variation points described. Moreover, if the product requires an additional feature there has to be an analysis of the introduction of this feature in the PL artifacts.

The fact that the metadata model makes explicit the relationships among artifacts enables the execution of algorithms of analysis to inspect the PL artifacts. However, we still lack a tool that makes this analysis possible.

The completion of this activity represents the end of an iteration of the variability management process, and thus a return to the PL development activity which triggered the beginning of the iteration.

4. Lessons Learned

In this section, we present the lessons learned from the development and the exercising of the variability management process.

Variability Management Process – the key issue of a PL is the way it articulates and manages its variable aspects. A variability management process must coexist with any PL core asset development and product development in order to support the clear specification, tracing, and control of variabilities. Our studies provided evidence that the variability management process enables the identification, and therefore the control, of more variabilities than the existing PL, as it establishes a sequence of activities, notation, guidelines, and support models and artifacts. Moreover, it makes explicit the most important decisions, such as the number of variants associated with a variation point and the type of choices allowed; the binding time; and the implementation mechanisms.

Empirical Evaluation of a PL – the evaluation of the proposed variability management process was carried out based on an empirical study in the form of a quantitative case study [24, 26] which enabled a clear demonstration of the need of a variability management process. This case study together with the results of previous work [6] made it possible to create an experimental baseline for a PL. However, investigation has to proceed to increase the volume of data available for experiments. An experimental basis for a PL can be built that allow inference of quality attributes of the PL itself and of potential products.

UML Models – the proposed process is based on the UML models. It extends these models by adding stereotypes to represent variability in the feature, use case, static type, and component models. Most of the stereotypes used in our process were inherited from previous work [15, 27, 28, 29]. In our process, we introduced the idea of using UML notes to make explicit the multiplicity and binding time associated with variation points. One of the advantages of using notes is that, because the notes belong to the UML meta-models, they can be read from any commercial tool that supports UML modelling.

Feature Modelling – In previous work [6], we did not use the feature model because we gave priority to modifying a traditional software process, to reduce the impact of a PL adoption. The feature model is not, as yet, a familiar artifact from the software engineers point of view. However, we introduced the feature model in our variability management process because it proved to be important from both the reuse and variability tracing perspectives. Moreover, the introduction of the feature model made our PL approach compatible with most existing approaches [14, 29, 30]. In particular, it is important for a PL to keep a clear relationship between the feature model and the PL's architecture [10].

Metadata Model – the metadata model was an important result of our process design. It provides information about the relationships between artifacts, thus allowing navigation throughout them. Moreover, it forms a basis for building a tool to automate the variability management process. We note that there is still few tools to support the PL approach, such as Ménage [31], Holmes [32] and pure:variants [30]. The proposed metadata model can be represented in XML to facilitate exchange of data with other PL support tools. IBM/Rational Requisite Pro and Rose have shown to be potentially feasible to support variability management. Requisite Pro supports the specification of the relationship between use cases and features. However, it still does not allow graphical representation of the feature model and it is not well integrated with Rose models. Rose allows the manipulation of the UML models to represent variability, and already provides some mechanisms to support artifact tracing. We believe

that the harmonization of tools such as the IBM/Rational suite with PL principles can be fruitful and should facilitate PL adoption.

5. Related Work

The PL variability management process proposed in this paper took into account previous work on PL approaches mainly related to the following issues: management activities, artifact notation, variability attributes, metadata modelling and experimental software engineering.

The management activities defined for the proposed process is inline with the essential activities of the SEI PLP [1] because variability management is considered a subprocess of the PL management activity. Thus, it has a close interaction with the core asset development and the product development activities.

The activities defined in our process were initially based on van Gurp, Bosch and Svahnberg's [5] variability management activities. However, their activities were not fully described. Our activities and their associated roles and artifacts are fully specified.

The proposed process uses UML as the notation in which to represent the PL artifacts. It took into consideration similar approaches such as Gomaa and Weber [16] (further extended in [29]), Kobra [14], and Clauß [15, 27]. These approaches extend the UML notation to represent variability aspects. In our process, only the use case, static type, and component models are traced as we considered these models as the most important in variability representation. However, we agree that the representation of variability in a sequence diagram, as in Gomaa [29], should also be considered to analyze the impact of variability on interactions.

In order to define the activities of the process it was important to consider previous attempts to define variability attributes [5, 19, 21, 22, 27]. On the end, we define the following attributes for variability: type, binding time and implementation mechanisms. Recent work presented by Czarnecki [11] and Cechticky [33] contains well-founded definitions of feature attributes and representation. However, they are not based on UML.

The metadata model proposed to support our process was based on that of Becker [22]. However, we have eliminated elements like those which represent the product family (ProductFamily and FamilyMember) and those which represent the variability implementation mechanism (Selection, Generation and Substitution). Morever, we added the following elements to support the variability tracing overall the PL artifacts: `VariantArtifact`, `FeatureModel`, `Feature`, `UseCaseModel`, `UseCase`, `TracingModel` and `TracingElement`.

Existing experimental works were used as the basis to undertake the case study. In particular, Basili [34] discusses experiment planning and Kitchenham [24, 26] proposes a terminology, the steps to undertake an experiment, and how to analyze the results.

6. Conclusions

The variability management process is one of the most important activities of core asset development and evolution. It is the variability management that enables the clear identification and tracing of the differences amongst products of a PL.

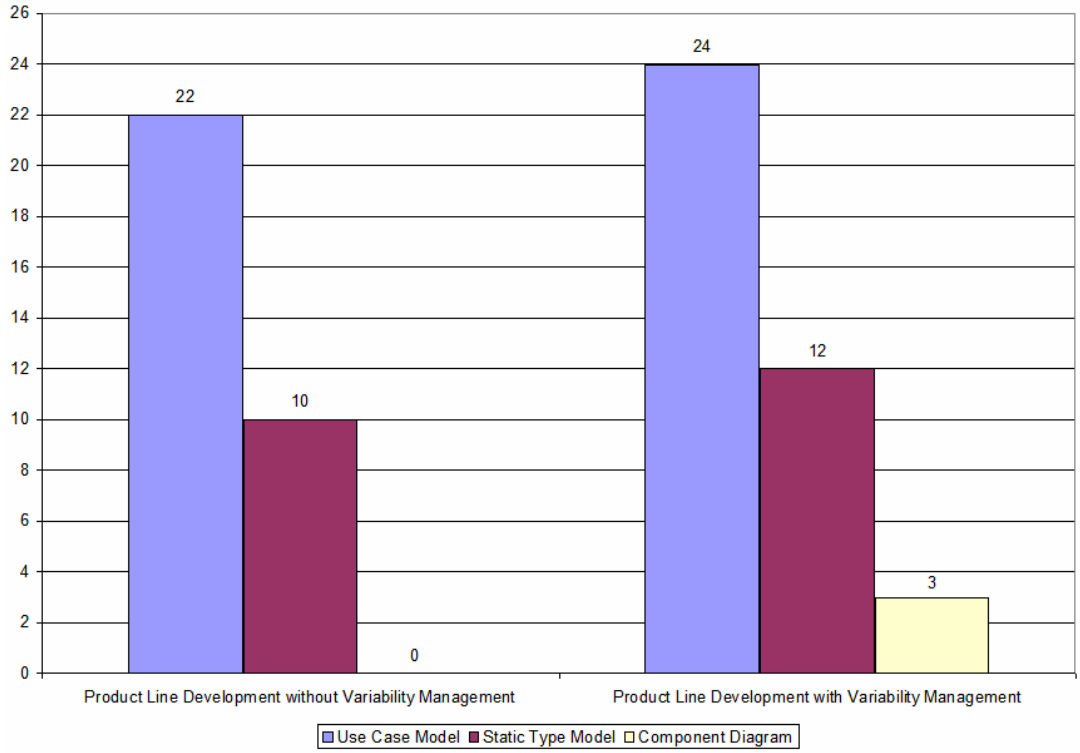

Figure 6: Number of variabilities identified for the use case, static type and component models.

This paper proposes a variability management process that makes explicit the activities, artefacts, and roles necessary to control variability in a PL approach that is based on UML. The proposed process was evaluated with a case study that compares a PL approach to the same with the introduction of variability management. Figure 6 shows the number of variabilities identified for the use case, static type and component models both for the original PL and after the introduction of the variability management. Thus, the results show the benefits of establishing well-defined and controlled variability management, as one of the main activities of a PL core asset development.

Moreover, a metadata model was proposed that forms the basis to design a tool to support variability management. We believe that automated support for the variability management process is an important issue to be investigated. It will make automated product configuration analysis possible, thus, providing the organizations with mechanisms to evaluate PL adoption and evolution. The experimental studies undertaken also need to be extended as they can provide a valuable basis with which to evaluate PLs.

Because the proposed process is triggered from the PL core asset development activities, it can be easily adapted to also PL approaches that are based on UML.

Acknowledgements

Itana M. S. Gimenes would like to thank the Computer System Group of the University of Waterloo, Canada, where she is currently in a sabbatical license.

Edson Alves de Oliveira Junior would like to thank the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES - Brazil) for financial support in his Master Degree project.

About the Authors

Edson Alves de Oliveira Junior has a bachelor and master degree in computer science from Universidade Estadual de Maringá, Brazil. His research interests include: software product line, software archi-

tecture, empirical software engineering, component-based development, and workflow management systems. He is currently taking doctoral degree in computer science at Universidade de São Paulo (ICMC-USP), Brazil.

Itana M. S. Gimenes is full professor of Software Engineering at Universidade Estadual de Maringá (DIN-UEM), Brazil, and Ph.D. from The University of York, Department of Computer Science, UK. Her research interests include: software architecture, software product line, component-based development, and workflow management systems. She is currently in a sabbatical license at CSG/SCS/University of Waterloo.

Elisa H. M. Huzita is associate professor at the Departamento de Informática of the Universidade Estadual de Maringá (DIN-UEM), Brazil. Her research interests include: component-based development and distributed software engineering.

José Carlos Maldonado is full professor at the Instituto de Ciências Matemáticas e de Computação (ICMC-USP) of the Universidade de São Paulo, Brazil. He is vice-president of the Brazilian Computer Society, and member of the ACM and IEEE.

References

[1] SEI - Software Engineering Institute. A framework for software product line practice 4.2. Pittsburgh. <http://www.sei.cmu.edu/productlines/framework.html >. Access: June, 01 2005.

[2] P. Heymans, J. C. Trigaux. Software product line: state of the art. Technical report for PLENTY project, Institut d'Informatique FUNDP, Namur, 2003.

[3] L. Bass, P. Clements, R. Kazman. Software architecture in practice. 2. ed. Boston: Addison-Wesley, 2003. 560 p.

[4] F. van der Linden. Software Product families in Europe: The Esaps and Café Projects, IEEE Software, July/August 2002, pp. 41-49.

[5] J. van Gurp, J. Bosch, M. Svahnberg. On the notion of variability in software product lines. in: THE WORKING IEEE/IFIP CONFERENCE ON SOFTWARE ARCHITECTURE (WICSA), 2001, Amsterdam. *Proceedings*… Amsterdam, 2001. pp. 45-54.

[6] I. M. S. Gimenes, E. A. Oliveira Junior, F. R. Lazilha, L. M. Barroca. A product line architecture for workflow management systems with component-based development, in: 2003 Proc. The IEEE Conference on Information Reuse and Integration, pp. 112-119.

[7] WfMC – Workflow Management Coalition. <http://www.wfmc.org>. Access: June, 10 2004.

[8] K. Kang. Feature-oriented domain analysis (FODA) - feasibility study. Technical Report CMU/SEI-90-TR-21, SEI/CMU, Pittsburgh, 1990.

[9] M. Simons, D. Creps, C. Klingler, L. Levine, D. Allemang. Organization domain modeling (ODM) guidebook, version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defence Systems, 1996.

[10] P. Sochos, I. Philippow, M. Riebish. Feature-oriented development of software product lines: mapping feature models to the architecture. Springer, LNCS 3263, 2004, pp. 138-152.

[11] K. Czarnecki, S. Helsen, U. Eisenecker. U. Staged configuration through specialization and multi-level configuration of feature models. To appear in special issue on "Software Variability: Process and Management," Software Process Improvement and Practice, 10(2), 2005.

[12] K. Czarnecki, U. Eisenecker, U., Generative programming. methods, tools, and applications. Addison-Wesley, 2000. 832 p.

[13] D. Batory. The Road to Utopia: A future for generative programming, in: Domain Specific Generation, Lengauer et al. (eds.), LNCS 3016, pp. 1-18, 2004.

[14] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthing, B. Paech, J. Wüst, J. Zeitel. Component-based product-line engineering with UML. Boston: Addison-Wesley, 2001.

[15] M. Clauß, Modeling variability with UML. in: YOUNG RESEARCHES WORKSHOP, 2001, Erfurt. Proceedings… Erfurt. 2001.

[16] H. Gomaa, D. Webber. Modeling adaptive and evolvable software product lines using the variation point model. in: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 37., 2004, Hawaii. Proceedings… Hawaii, 2004. pp. 01-10.

[17] I. Jacobson, M. Griss, P. Jonsson. Software reuse - architecture process and organization for business success. 1. ed. Boston: Addison-Wesley, 1997. 528 p.

[18] M. Morisio, G. H. Travassos, M. E. Stark. Extending UML to support domain analysis. in: THE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 5., 2000, Grenoble, France. pp. 321-324.

[19] J. van Gurp, J. Bosch. Managing variability in software product lines. in: Proceedings of the Landelijk Architectuur Congres. Amsterdam, 2000.
[20] C. Fritsch, A. Lehn, T. Strohm. Evaluating variability implementation mechanisms. in: INTERNATIONAL WORKSHOP ON PRODUCT LINE ENGINEERING, 2., 2002, Seattle, USA. 2002. pp. 59-64.
[21] M. Anastasopoulos, C. Gracek. Implementing product line variabilities. in: ACM SIGSOFT Software Engineering Notes, New York, v. 26, n. 3, pp. 109-117, May. 2001.
[22] M. Becker. Towards a general model of variability in product families. in: SOFTWARE VARIABILITY MANAGEMENT WORKSHOP, 2003, Portland. pp. 19-27.
[23] IBM Rational Software - <http://www.ibm.com/software/rational> - Access: Nov. 2004.
[24] B. Kitchenham, L. Pickard, S. L. Pfleeger. Case studies for method and tool evaluation. IEEE Software, v.11, pp. 52-62, 1995.
[25] S. L. Pfleeger. Experimental design and analysis in software engineering – how to set up an experiment. ACM SIGSOFT – software Engineering Notes. v. 20, n. 1, pp. 22-26, 1995.
[26] B. Kitchenham. DESMET: a method for evaluating software engineering methods and tools. Technical Report TR96-09, Keele, United Kingdom, 1996. 49 p.
[27] M. Clauß. Generic modeling using UML extensions for variability. in: OOPSLA 2001 WORKSHOP ON DOMAIN SPECIFIC VISUAL LANGUAGES, 1. 2001, Tampa Bay. USA, pp. 11-18.
[28] H. Gomaa, M. E. Shin. Multiple-view meta-modeling of software product lines, 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society 2002, ISBN 0-7695-1757-9, pp. 238-246, 2002.
[29] H. Gomaa. Designing software product lines with UML: from use cases to pattern-based software architectures. Boston: Addison-Wesley, 2005.
[30] PURE-SYSTEMS - pure-variants: Variant Management – <http://web.pure-systems.com/Variant_Management.49.0.html> - Access: Nov. 2004.
[31] A. van der Hoek. Capturing product line architectures. In: INTERNATIONAL SOFTWARE ARCHITECTURE WORKSHOP, 4., 2000, Limerick. Proceedings… Limerick, 2000. pp. 95-99.
[32] G. Succi, J. Yip, W. Pedrycz. Holmes: an intelligent system to support software product line Development. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 23., 2001, Toronto, Canada. pp. 829-832.
[33] Cechticky, V., Passetti, A., Rohlik, O., Schaufelberger, W., XML-based feature modelling. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, Madrid, Spain, pp. 101-114, LNCS 3107, Jul. 2004.
[34] V. R. Basili, R. W. Selby, D. H. Hutchens. Experimentation in software engineering. IEEE Transactions on Software Engineering, Piscataway, v. 12, n. 7, pp. 733-743, 1986.
[35] M. Svahnberg; J. Van Gurp; J. Bosch. A taxonomy of variability realization techniques. Technical report, Blekinge Institute of Technology, Sweden, 2002.

Appendix A:

UML Relations and Stereotypes Used in the Graphical Representation of the Variation Points and Variants

	Use Case Diagram		Class Diagram		Component Diagram	
	UML Relation	**Element Stereotype**	**UML Relation**	**Element Stereotype**	**UML Relation**	**Element Stereotype**
Variation Point	--------	<<variationPoint>>	--------	<<variationPoint>>	Dependency	<<variable>>
Mandatory Variant	Association Aggregation Composition Dependency	<<mandatory>>	Association Aggregation Composition Dependency	<<mandatory>>	Dependency	<<mandatory>>
Optional Variant	Association Aggregation Composition Dependency	<<optional>>	Association Aggregation Composition Dependency	<<optional>>	Dependency	<<optional>>
Inclusive Alternative Variant	Spec./ General. with the stereotype <<extend>>	<<alternative_OR>>	Inheritance	<<alternative_OR>>	Dependency	<<alternative_OR>>
Exclusive Alternative Variant	Spec./ General. with the stereo-type <<extend>>	<<alternative_XOR>>	Inheritance	<<alternative_XOR>>	Dependency	<<alternative_XOR>>
Mutually Exclusive Variant	Dependency	<<mutex>>	Dependency	<<mutex>>	Dependency	<<mutex>>
Inclusive Variant	Dependency	<<requires>>	Dependency	<<requires>>	Dependency	<<requires>>